1

Synthesis of Composition and Discrimination Operators for Divide-and-Conquer Logic Programs

PIERRE FLENER, YVES DEVILLE

This chapter gives a brief overview of our framework for stepwise synthesis of logic programs from examples and properties. Directives are extracted for the development of a particular synthesis mechanism whose steps are guided by a divide-and-conquer schema. It features deductive and inductive reasoning. Examples and properties are presented to it in a non-incremental fashion. The objectives and methods of its last steps (synthesis of composition and discrimination operators) are formalized, and illustrated on some sample problems.

This chapter is organized as follows. After the introduction, three sample problems are presented in section 1.2. Sections 1.3 to 1.5 present the objectives and methods of some synthesis steps, and illustrate them on the sample problems. Some conclusions on the results are drawn in section 1.6, related work is stated, and future research directions are outlined.

1.1 INTRODUCTION

Program synthesis research [Bie92] aims at automating the passage from specifications to programs, in opposition to more traditional, mostly manual, programming techniques. The key question here is: "what is a specification?". Today, an emerging

Constructing Logic Programs, J.-M. Jacquet (editor)

consensus is that one may speak of *synthesis* if the specification does not explicitly reveal recursion or iteration. Otherwise, the technique could be classified as *transformation*. In this introductory section, we present a brief overview of our framework for the stepwise synthesis of logic programs from examples and properties of the relation to be implemented. This framework is developed in more detail in [Fle93], [FD91], and [FD93].

1.1.1 Specifications by examples and properties

We first define a possible starting point of synthesis.

Definition 1 *A* specification by examples and properties *of a procedure r/n consists of:*

i) a set $E(r)$ of ground examples of the behavior of r;
ii) a set $P(r)$ of properties (first-order logic statements) of r.

Examples are a very appealing means of conveying information about a relation: they are easy to elaborate or understand, and they implicitly reveal some manipulations of the parameters. However, examples alone constitute an incomplete information source, and they lack in expressive power. We believe the specifier knows the missing details, and should thus be able to provide them, rather than have the synthesis mechanism guess them. We depart thus from traditional example-based synthesis and allow a property set in our specifications, where properties are meant to overcome the drawbacks of examples, and yet have the same appeal as examples. Non-recursive Horn clauses are a very convenient format. (Since we assume here that synthesis starts from a non-recursive specification, we do not consider recursive properties.) Sample sets of examples and properties appear in section 1.2.

1.1.2 Logic algorithms

We are actually only interested in synthesizing algorithms, rather than full-fledged programs. Indeed, algorithm design in itself is already very hard, and we do not want to encumber ourselves with the additional burdens of algorithm optimization, transformation, and implementation, which are well-researched topics anyway ([Dev90]). Algorithms expressed in a logic formalism are here called logic algorithms ([Dev90]).

Definition 2 *A* logic algorithm *of a procedure r, denoted $LA(r)$, consists of a formula of the form: $r(X,Y) \Leftrightarrow Def[X,Y]$, where the body Def is a first-order logic statement.*[1]

Executable Prolog programs can be easily derived from logic algorithms with bodies in disjunctive normal form [Dev90]. Sample logic algorithms appear in section 1.2.

[1] Extending the syntactical conventions stated in section ??, (predicate) variable names start with an uppercase; functors and predicate names start with a lowercase. The string $F[X,Y]$ denotes a formula F whose free variables are X and Y; the string $F[a,b]$ denotes $F[X,Y]$ where the free occurrences of X and Y have been replaced by the terms a and b, respectively. The variables X and Y are assumed to be universally quantified over $LA(r)$; other free variables in Def are assumed to be existentially quantified over Def.

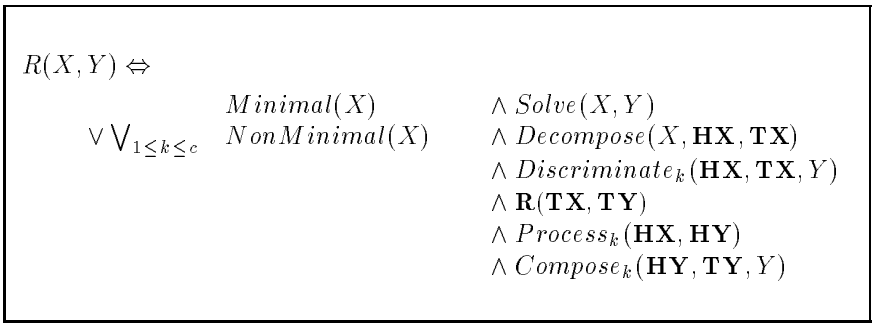

Figure 1.1 The divide-and-conquer logic algorithm schema

1.1.3 Schema-guided design of logic algorithms

Algorithm schemata are an old idea of computer science (see an early survey in [Man74]). They are template algorithms with fixed control flows. They embody the essence of algorithm design strategies (such as divide-and-conquer, generate-and-test, global search, ...) and are thus an invaluable knowledge source for guiding (semi-) automated algorithm design.

Example 1 Loosely speaking, a *divide-and-conquer* algorithm for a binary predicate r over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is found by directly solving the problem. Otherwise, i.e. if X is non-minimal, we decompose X into a series[2] $\mathbf{HX}$ of heads of X and a series $\mathbf{TX}$ of tails of X, the latter being of the same type as X, as well as smaller than X according to some well-founded relation. The tails $\mathbf{TX}$ recursively yield tails $\mathbf{TY}$ of Y. The heads $\mathbf{HX}$ are processed into a series $\mathbf{HY}$ of heads of Y. Finally, Y is composed from its heads $\mathbf{HY}$ and tails $\mathbf{TY}$. It may happen that different processing and composition operators emerge for the non-minimal form of X: we have to discriminate between them according to the values of $\mathbf{HX}$, $\mathbf{TX}$, and Y. Non-determinism of the intended relation results in some discriminants that are always *true*. ◇

Logic algorithm schemata can be expressed as second-order logic algorithms. For instance, many logic algorithms designed by a divide-and-conquer strategy, and having a single minimal case and a single non-minimal case, fit the schema of figure 1.1, where $\mathbf{R}(\mathbf{TX}, \mathbf{TY})$ stands for an optional conjunction of recursive atoms. To simplify the presentation of this chapter, the proposed examples will be based on the schema of figure 1.2. This is a particular case of figure 1.1, where the induction parameter X is a list, decomposed in the traditional head/tail form. In this particular case, the non-minimal test and the decomposition operator could have been merged. We also assume that the processing and composition operators are handled by a single predicate.

[2] In a logic algorithm schema, a bold term (resp. atom) denotes a vector (resp. conjunction) of terms (resp. atoms).

$$
\begin{array}{l}
R(X,Y) \Leftrightarrow \\
\qquad\qquad X = [\,] \qquad\quad \wedge\, Solve(X,Y) \\
\quad \vee \bigvee_{1 \le k \le c} \; X = [\_|\_] \quad \wedge\, X = [HX \mid TX] \\
\qquad\qquad\qquad\quad \wedge\, Discriminate_k(HX, TX, Y) \\
\qquad\qquad\qquad\quad \wedge\, R(TX, TY) \\
\qquad\qquad\qquad\quad \wedge\, ProcComp_k(HX, TY, Y)
\end{array}
$$

Figure 1.2 The divide-and-conquer logic algorithm schema for lists

1.1.4 Directives for the development of a synthesis mechanism

We have developed in [FD91, FD93] a general strategy for stepwise, progressive, consistent, and sound synthesis of logic algorithms from specifications by examples and properties. In stepwise synthesis, there is a series of refinements towards a correct logic algorithm:

$$LA_1(r), LA_2(r), \cdots, LA_i(r), \cdots, LA_f(r)$$

At each step, we measure the current logic algorithm against the intended relation: correctness criteria useful for characterizing the soundness of synthesis have been identified. Across several steps, we measure the progression of the synthesized logic algorithms towards the intended relation: comparison criteria useful for characterizing the progression of synthesis have also been identified. These criteria may be used during the development of a synthesis mechanism to ensure a sound synthesis.

This general strategy can be particularized to schema-guided synthesis, where each step instantiates some predicate variable(s) of a schema. We adopt the divide-and-conquer schema of figure 1.1 for guiding synthesis. Indeed, the class of algorithms that can be designed by this strategy is fairly large and important. There are eight steps to our mechanism [Fle91]:

Step 1: Syntactic creation of a first approximation
Step 2: Synthesis of $Minimal$ and $NonMinimal$
Step 3: Synthesis of $Decompose$
Step 4: Introduction of the recursive atoms
Step 5: Synthesis of $Solve$
Step 6: Synthesis of the $Process_k$ and $Compose_k$
Step 7: Synthesis of the $Discriminate_k$
Step 8: Syntactic generalization

Steps 1 to 3 are straightforward, and are based on type knowledge. Step 4 is relatively easy, and is performed by deductive reasoning from the property set. Step 5 is a particular case of steps 6 and 7. The latter are the real challenges of synthesis, and are the main topics of this paper. Step 8 is straightforward again, but is also covered here. The development of all these steps can be found in [Fle93].

Note that the divide-and-conquer schema is not an input to the synthesis mechanism, but rather hardwired into it.

We implement deterministic and non-deterministic relations using the same synthesis mechanism. The choice of an induction parameter is a priori independent of the way (*mode*) the resulting program can be used.

Examples give rise to inductive synthesis (generalization ([Sum77, BS79, Bie84]), learning ([Sha82, Tin90, Mug92])), whereas axiomatic specifications (and thus properties) give rise to deductive synthesis (proofs-as-programs ([MW79, BSW90, Fri90]), transformations ([Han80, Cla81, Hog81, LP90])). Since we have both kinds of specification information, we want to avoid using only one kind of inference, and thus degrading the non-used information source into validation information. We therefore strive for inductive *and* deductive synthesis, using whichever inference kind is best suited at each step. This approach of course precludes synthesis if only examples or only properties are given. It also gives a constructive role to each kind of specification information.

There are two ways of presenting examples (and properties): *one-by-one (incrementally)*, or *all-at-once*. The former approach, advocated in [Sha82] and by the Inductive Logic Programming (ILP) school of thought [Mug92], has some nice convergence properties. But we adhere to the school of thought advocated in [Sum77], where the examples are presented all-at-once, so that a maximum of information is available at each step.

With example-based synthesis, constants from $E(r)$ inevitably appear in the logic algorithms, thus destroying completeness unless they are generalized. We thus need one more notion:

Definition 3 *Let* Γ *be a total function in the set of logic algorithms, such that* $\Gamma(LA(r))$ *is* $LA(r)$ *without its equality atoms involving constants introduced from* $E(r)$.

This is illustrated in subsequent sections. It can be shown that Γ is a generalization function, i.e. that $\Gamma(LA(r))$ is at least as general as $LA(r)$. Informally speaking, a logic algorithm is more general than another one iff its body is more often *true* than the body of the other one.

The rest of this chapter is organized as follows. In section 1.2, we present three sample problems. In sections 1.3 to 1.5, we present the objectives and methods of steps 6 to 8, respectively, and illustrate them on the sample problems. In section 1.6, we draw some conclusions on our results, state some related work, and outline future research directions.

1.2 SAMPLE PROBLEMS

In this section, we state three sample problems that are used throughout the remainder of this chapter. The problems can be studied independently, both here and in subsequent sections. But they are complementary in the sense that they illustrate distinct intricacies of logic algorithm synthesis. Each problem is first specified informally, then by examples and properties. These examples and properties are supposed to be given: this chapter does not discuss the elaboration of such specifications. For each problem, a sample logic algorithm (constructed by the methodology of [Dev90]) is shown in order to illustrate what the synthesis mechanism should achieve. For each problem,

$E(efface)$ = {
`efface(a,[a],[])` (EE_1)
`efface(b,[b,c],[c])` (EE_2)
`efface(e,[d,e],[d])` (EE_3)
`efface(f,[f,g,h],[g,h])` (EE_4)
`efface(j,[i,j,k],[i,k])` (EE_5)
`efface(p,[m,n,p],[m,n])` } (EE_6)

$P(efface)$ = {
`efface(X,[X|T],T)` (PE_1)
`efface(X,[Y,X|T],[Y|T])` $\Leftarrow$ `X` $\neq$ `Y` } (PE_2)

Figure 1.3 Sample versions of $E(efface)$ and $P(efface)$

$$
\begin{array}{l}
efface(E, L, R) \Leftrightarrow \\
\qquad L = [HL|TL] \quad \wedge\ E = HL\ \wedge\ R = TL \\
\vee \quad L = [HL|TL] \quad \wedge\ E \neq HL \\
\qquad\qquad\quad \wedge\ efface(E, TL, TR) \\
\qquad\qquad\quad \wedge\ R = [HL|TR]
\end{array}
$$

Figure 1.4 A sample version of $LA(efface)$

we also give the logic algorithm synthesized by step 5, but, again, we do not discuss how this is performed.

1.2.1 The $efface/3$ problem

The `efface(E,L,R)` procedure succeeds iff term `E` belongs (at least once) to the non-empty list `L`, and list `R` is `L` without its first occurrence of `E`.

Figure 1.3 gives sample versions of $E(efface)$ and $P(efface)$. Note that properties PE_1 and PE_2 generalize the examples of $\{EE_1, EE_2, EE_4\}$ and $\{EE_3, EE_5\}$, respectively.

Figure 1.4 gives a sample version of $LA(efface)$.

Figure 1.5 shows $LA_5(efface)$, where disjunct DE_i corresponds to example EE_i. We assume that step 2 chose L as induction parameter, and introduced one minimal case (disjunct DE_1, where L has exactly one element), and one non-minimal case (disjuncts DE_2 to DE_6, where L has at least two elements). Step 3 decomposed the non-minimal form into its head HL and tail TL. Step 4 introduced recursive atoms into some non-minimal disjuncts. Note that E was judged to be an *auxiliary parameter* (because it is of a non-inductive type[3]): this prevented the search for a tail TE of E while introducing recursive atoms; moreover, this prevented a superfluous introduction

[3] Inductive types are here assumed to be either integers or lists.

$$
\begin{array}{lll}
\textbf{efface}(\mathbf{E},\mathbf{L},\mathbf{R}) \Leftrightarrow & & \\
\quad \mathbf{L} = [\_] & \wedge\ \mathbf{L} = [\mathbf{E}]\ \wedge\ \mathbf{R} = [\,] & \\
 & \wedge\ E = a\ \wedge\ L = [a]\ \wedge\ R = [\,] & (DE_1) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \mathbf{E} = \mathbf{HL}\ \wedge\ \mathbf{R} = \mathbf{TL} & \\
 & \wedge\ E = b\ \wedge\ L = [b,c]\ \wedge\ R = [c] & \\
 & \wedge\ HL = b\ \wedge\ TL = [c] & (DE_2) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \textbf{efface}(\mathbf{E},\mathbf{TL},\mathbf{TR}) & \\
 & \wedge\ E = e\ \wedge\ L = [d,e]\ \wedge\ R = [d] & \\
 & \wedge\ HL = d\ \wedge\ TL = [e]\ \wedge\ TR = [\,] & (DE_3) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \mathbf{E} = \mathbf{HL}\ \wedge\ \mathbf{R} = \mathbf{TL} & \\
 & \wedge\ E = f\ \wedge\ L = [f,g,h]\ \wedge\ R = [g,h] & \\
 & \wedge\ HL = f\ \wedge\ TL = [g,h] & (DE_4) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \textbf{efface}(\mathbf{E},\mathbf{TL},\mathbf{TR}) & \\
 & \wedge\ E = j\ \wedge\ L = [i,j,k]\ \wedge\ R = [i,k] & \\
 & \wedge\ HL = i\ \wedge\ TL = [j,k]\ \wedge\ TR = [k] & (DE_5) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \textbf{efface}(\mathbf{E},\mathbf{TL},\mathbf{TR}) & \\
 & \wedge\ E = p\ \wedge\ L = [m,n,p]\ \wedge\ R = [m,n] & \\
 & \wedge\ HL = m\ \wedge\ TL = [n,p]\ \wedge\ TR = [n] & (DE_6)
\end{array}
$$

Figure 1.5 $LA_5(efface)$

of recursion in disjuncts DE_2 and DE_4. Step 5 solved the minimal case and the non-recursive, non-minimal case. The atoms in boldface represent $\Gamma(LA_5(efface))$.

1.2.2 The $perm/2$ problem

The `perm(L,P)` procedure succeeds iff list `P` is a permutation of list `L`.

Figure 1.6 gives sample versions of $E(perm)$ and $P(perm)$. Note that properties PP_1 to PP_3 generalize examples EP_2 to EP_4, respectively.

Figure 1.7 gives a sample version of $LA(perm)$.

Figure 1.8 shows $LA_5(perm)$, where disjunct DP_i corresponds to example EP_i. We assume that step 2 chose L as induction parameter, and introduced one minimal case (disjunct DP_1, where L is empty), and one non-minimal case (disjuncts DP_2 to DP_{10}, where L has at least one element). Step 3 decomposed the non-minimal form into its head HL and tail TL. Step 4 introduced a recursive atom into each non-minimal disjunct: note that the values of TP (the tail of P) could not have been computed

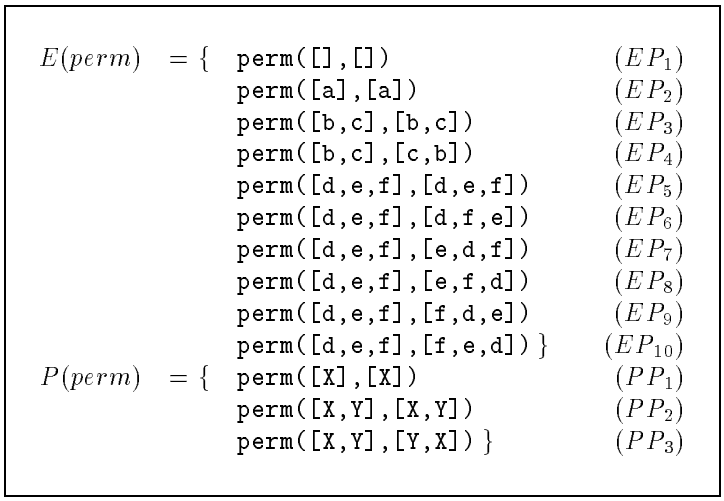

Figure 1.6 Sample versions of $E(perm)$ and $P(perm)$

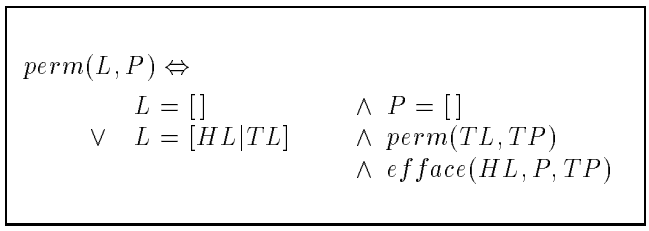

Figure 1.7 A sample version of $LA(perm)$

deterministically in disjuncts $DP5$ to DP_{10}. Step 5 solved the minimal case. The atoms in boldface represent $\Gamma(LA_5(perm))$.[4]

$$
\begin{array}{lll}
\mathbf{perm(L,P)} \Leftrightarrow & & \\
\quad & \mathbf{L} = [\,] & \wedge\ \mathbf{P} = [\,] \\
\quad & & \wedge\ L = [\,] \wedge\ P = [\,] \qquad (DP_1) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [a] \wedge\ P = [a] \\
\quad & & \wedge\ HL = a \wedge\ TL = [\,] \wedge\ TP = [\,] \qquad (DP_2) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [b,c] \wedge\ P = [b,c] \\
\quad & & \wedge\ HL = b \wedge\ TL = [c] \wedge\ TP = [c] \qquad (DP_3) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [b,c] \wedge\ P = [c,b] \\
\quad & & \wedge\ HL = b \wedge\ TL = [c] \wedge\ TP = [c] \qquad (DP_4) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_5) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [d,e,f] \wedge\ P = [d,f,e] \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_6) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [d,e,f] \wedge\ P = [e,d,f] \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_7) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [d,e,f] \wedge\ P = [e,f,d] \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_8) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [d,e,f] \wedge\ P = [f,d,e] \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_9) \\
\vee & \mathbf{L} = [\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \wedge\ \mathbf{perm(TL,TP)} \\
\quad & & \wedge\ L = [d,e,f] \wedge\ P = [f,e,d] \\
\quad & & \wedge\ HL = d \wedge\ TL = [e,f] \wedge\ TP \in \{[e,f],[f,e]\} \qquad (DP_{10})
\end{array}
$$

Figure 1.8 $LA_5(perm)$

1.2.3 The $firstSeq/3$ problem

The `firstSeq(L,F,S)` procedure succeeds iff list `F` is the first maximal sequence of identical elements at the beginning of list `L`, and list `S` is the corresponding suffix of `L`.

[4] The $E \in S$ predicate holds iff term E belongs to the set S. Note that, in a logic algorithm, boldface terms and atoms do not denote vectors and conjunctions, contrary to logic algorithm schemas.

$E(firstSeq)$ = {
`firstSeq([a],[a],[])` (EF_1)
`firstSeq([b,c,d,e],[b],[c,d,e])` (EF_2)
`firstSeq([f,f,g,h],[f,f],[g,h])` (EF_3)
`firstSeq([i,i,i,j],[i,i,i],[j])` } (EF_4)

$P(firstSeq)$ = {
`firstSeq([X],[X],[])` (PF_1)
`firstSeq([X,X],[X,X],[])` (PF_2)
`firstSeq([X,Y|T],[X],[Y|T])` $\Leftarrow$ `X` $\neq$ `Y` } (PF_3)

Figure 1.9 Sample versions of $E(firstSeq)$ and $P(firstSeq)$

Figure 1.9 gives sample versions of $E(firstSeq)$ and $P(firstSeq)$. Note that properties PF_1 and PF_3 generalize examples EF_1 and EF_2, respectively.

Figure 1.10 gives a sample version of $LA(firstSeq)$.

$$
\begin{array}{l}
firstSeq(L,F,S) \Leftrightarrow \\
\quad L = [HL] \qquad\qquad \wedge\ F = L\ \wedge\ S = [\,] \\
\vee\ \ L = [HL_1, HL_2|TL] \quad \wedge\ HL_1 \neq HL_2 \wedge F = [HL_1] \wedge S = [HL_2|TL] \\
\vee\ \ L = [HL_1, HL_2|TL] \quad \wedge\ HL_1 = HL_2 \\
\qquad\qquad\qquad\qquad\ \ \wedge\ firstSeq([HL_2|TL], TF, TS) \\
\qquad\qquad\qquad\qquad\ \ \wedge\ F = [HL_1|TF]\ \wedge\ S = TS
\end{array}
$$

Figure 1.10 A sample version of $LA(firstSeq)$

Figure 1.11 shows $LA_5(firstSeq)$, where disjunct DF_i corresponds to example EF_i. We assume that step 2 chose L as induction parameter, and introduced one minimal case (disjunct DF_1, where L has exactly one element), and one non-minimal case (disjuncts DF_2 to DF_4, where L has at least two elements). Step 3 decomposed the non-minimal form into its head HL and tail TL. Step 4 introduced a recursive atom into the last two non-minimal disjuncts only: indeed, recursion would be useless in the first non-minimal disjunct where F and S can already be computed directly from HL and TL. Step 5 solved the minimal case and the non-recursive, non-minimal case. The atoms in boldface represent $\Gamma(LA_5(firstSeq))$.

1.3 SYNTHESIS OF THE $PROCESS_K$ AND $COMPOSE_K$ (STEP 6)

In the non-minimal recursive case, the $Process_k(\mathbf{HX}, \mathbf{HY})$ procedures process the heads **HX** of the induction parameter X into the heads **HY** of the other parameter Y, in case X is non-minimal. The $Compose_k(\mathbf{HY}, \mathbf{TY}, Y)$ procedures compose

$$
\begin{array}{lll}
\textbf{firstSeq}(\mathbf{L},\mathbf{F},\mathbf{S}) \Leftrightarrow & & \\
\quad \mathbf{L} = [\_] & \wedge\ \mathbf{F} = \mathbf{L} \wedge\ \mathbf{S} = [\,] & \\
 & \wedge\ L = [a] \wedge\ F = [a] \wedge\ S = [\,] & (DF_1) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \mathbf{TL} = [\mathbf{HTL}|\_] \wedge\ \mathbf{HL} \neq \mathbf{HTL} & \\
 & \wedge\ \mathbf{F} = [\mathbf{HL}] \wedge\ \mathbf{S} = \mathbf{TL} & \\
 & \wedge\ L = [b,c,d,e] \wedge\ F = [b] \wedge\ S = [c,d,e] & \\
 & \wedge\ HL = b \wedge\ TL = [c,d,e] & (DF_2) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \mathbf{firstSeq}(\mathbf{TL},\mathbf{TF},\mathbf{TS}) & \\
 & \wedge\ L = [f,f,g,h] \wedge\ F = [f,f] \wedge\ S = [g,h] & \\
 & \wedge\ HL = f \wedge\ TL = [f,g,h] & \\
 & \wedge\ TF = [f] \wedge\ TS = [g,h] & (DF_3) \\
\vee\ \ \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] & \\
 & \wedge\ \mathbf{firstSeq}(\mathbf{TL},\mathbf{TF},\mathbf{TS}) & \\
 & \wedge\ L = [i,i,i,j] \wedge\ F = [i,i,i] \wedge\ S = [j] & \\
 & \wedge\ HL = i \wedge\ TL = [i,i,j] & \\
 & \wedge\ TF = [i,i] \wedge\ TS = [j] & (DF_4)
\end{array}
$$

Figure 1.11 $LA_5(firstSeq)$

parameter Y from its heads $\mathbf{HY}$ (obtained via processing $\mathbf{HX}$) and tails $\mathbf{TY}$ (obtained via recursion on $\mathbf{TX}$). We merge each $Process_k(\mathbf{HX}, \mathbf{HY})$ with its counterpart $Compose_k(\mathbf{HY}, \mathbf{TY}, Y)$ into $ProcComp_k(\mathbf{HX}, \mathbf{TY}, Y)$ so that their implementations are synthesized at the same time. In this section, we first formally present the objective and methods of step 6. Then we synthesize $ProcComp_k$ operators for the sample problems given in section 1.2.

1.3.1 Formalization: objective and methods

Given $LA_5(r)$ as follows:

$$
\begin{array}{lll}
r(X,Y) \Leftrightarrow & & \\
\quad & minimal(X) & \wedge\ solve(X,Y) \\
\quad & & \wedge\ \bigvee_{1\le i\le b} X = x_i\ \wedge\ Y = y_i \\
\quad \vee & nonMinimal(X) & \wedge\ decompose(X, \mathbf{HX}, \mathbf{TX}) \\
\quad & & \wedge\ \mathbf{r}(\mathbf{TX}, \mathbf{TY}) \\
\quad & & \wedge\ \bigvee_{b<i\le m} X = x_i\ \wedge\ Y = y_i \\
\quad & & \quad \wedge\ \mathbf{HX} = \mathbf{hx}_i\ \wedge\ \mathbf{TX} = \mathbf{tx}_i\ \wedge\ \mathbf{TY} \in \mathbf{ty}_i
\end{array}
$$

(where the last $m - b$ of the m examples are "covered" by the non-minimal disjuncts, and the first b examples are "covered" by the minimal disjuncts), the aim at step 6 is to transform $LA_5(r)$ into $LA_6(r)$ such that it fits the following schema:

$$
\begin{array}{lll}
r(X,Y) \Leftrightarrow & & \\
\quad & minimal(X) & \wedge\ solve(X,Y) \\
\quad & & \wedge\ \bigvee_{1\le i\le b} X = x_i \wedge Y = y_i \\
\quad \vee \bigvee_{1\le k\le c} & nonMinimal(X) & \wedge\ decompose(X, \mathbf{HX}, \mathbf{TX}) \\
\quad & & \wedge\ \mathbf{r}(\mathbf{TX}, \mathbf{TY}) \\
\quad & & \wedge\ ProcComp_k(\mathbf{HX}, \mathbf{TY}, Y) \\
\quad & & \wedge\ \bigvee_{i\in|k|} X = x_i \wedge Y = y_i \\
\quad & & \quad \wedge\ \mathbf{HX} = \mathbf{hx}_i\ \wedge\ \mathbf{TX} = \mathbf{tx}_i\ \wedge\ \mathbf{TY} = \mathbf{ty}'_i
\end{array}
$$

This amounts to partitioning the non-minimal recursive disjuncts into c equivalence classes (named $|1|, \ldots, |c|$) where the disjuncts of class $|k|$ have equal instantiations of $ProcComp_k$. Moreover, step 4 (introduction of the recursive atoms) produces several potential values of the parameters $\mathbf{TY}$ if the intended relation is non-deterministic: the second objective of step 6 is to trim these sets $\mathbf{ty}_i$ to singleton sets $\mathbf{ty}'_i$.

We have identified two methods to synthesize implementations of the $ProcComp_k$:

i) computation of most specific generalizations (msg, for short): the MSG Method applies if each $ProcComp_k$ is implemented as a conjunction of equality atoms;

ii) synthesis from an inferred specification by examples and properties: the Synthesis Method applies if each $ProcComp_k$ itself needs a full-fledged recursive logic algorithm, i.e. is implemented as a disjunction of conjunctions of literals.

We discuss these methods in turn.

1.3.1.1 THE MSG METHOD

The MSG Method constructs equivalence classes incrementally and non-deterministically.

Definition 4 *The* msg *of a set of disjuncts $\mathcal{D}$ is the msg of the $\langle \mathbf{hx}_i, \mathbf{ty}_i, y_i \rangle$ value-tuples extracted from the $\mathbf{HX} = \mathbf{hx}_i$, $\mathbf{TY} \in \mathbf{ty}_i$, and $Y = y_i$ atoms of the disjuncts $D_i \in \mathcal{D}$, if the $\mathbf{ty}_i$ are singletons, and undefined otherwise.*

Definition 5 *A disjunct D'_i is an* alternative *of disjunct D_i iff D'_i is obtained from D_i by non-deterministically trimming the sets $\mathbf{ty}_i$ of the $\mathbf{TY} \in \mathbf{ty}_i$ atoms to singletons $\mathbf{ty}'_i$.*

Definition 6 *An alternative D'_i of disjunct D_i is* compatible *with a set of disjuncts $\mathcal{S}$ iff the msg $\langle \mathbf{hx}, \mathbf{ty}, y \rangle$ between the $\langle \mathbf{hx}_i, \mathbf{ty'}_i, y_i \rangle$ value-tuple extracted from D'_i and the msg of $\mathcal{S}$ is such that y is a term whose variables are among the ones occurring in $\mathbf{hx}$ and $\mathbf{ty}$.*

Now, the algorithm goes as follows. Initially, there are no classes. At any moment, the non-minimal recursive disjuncts can be classified according to whether or not they belong to some equivalence class. Progression is achieved by selecting a disjunct D_i that doesn't belong to any class. If some alternative D'_i of D_i is compatible with some class $\mathcal{C}$, then $\mathcal{C}$ becomes $\mathcal{C} \cup \{D'_i\}$. Otherwise a new singleton class $\{D'_i\}$ is added, where D'_i is some alternative of D_i.

Once the equivalence classes have been computed, an assessing heuristic is applied:

Heuristic 1 *If there are more equivalence classes than non-minimal properties, then the msgs probably only cover the given examples, but not examples with larger parameters: invoke the Synthesis Method.*

Unless the Synthesis Method is judged to be applicable, the MSG Method continues. Let $procComp_k$ be the chosen instantiations of $ProcComp_k$. The msg $\langle \mathbf{hx}_k, \mathbf{ty}_k, y_k \rangle$ of class $|k|$ is rewritten as follows:

$$procComp_k(\mathbf{HX}, \mathbf{TY}, Y) \Leftrightarrow \mathbf{HX} = \mathbf{hx}_k \wedge \mathbf{TY} = \mathbf{ty}_k \wedge Y = y_k$$

so that it can be unfolded into the corresponding disjuncts.

1.3.1.2 THE SYNTHESIS METHOD

The Synthesis Method assumes that there is one single equivalence class, and that its $ProcComp(\mathbf{HX}, \mathbf{TY}, Y)$ is implemented as a possibly recursive disjunction of conjunctions of literals, i.e. it could be synthesized from scratch, just like any other logic algorithm. Let $procComp$ be the chosen instantiation of $ProcComp$. A specification by examples and properties for $procComp(\mathbf{HX}, \mathbf{TY}, Y)$ has to be inferred from $LA_5(r)$.

The inference of an example set is straightforward: just extract all the $\langle \mathbf{hx}_i, \mathbf{ty}_i, y_i \rangle$ tuples from the non-minimal disjuncts of $LA_5(r)$. If there are several alternatives, extract from the alternatives that led to compatibility during the MSG Method.

The inference of a property set is based on the observation that properties of the original problem are inherited by the subproblem. For every property of the form:

$$\texttt{r(x,y)} \Leftarrow \texttt{Body} \qquad (P_i)$$

find variants:

$$\texttt{r(tx,ty)} \qquad (E_j/P_j)$$

of examples or body-less properties, such that:

$$\texttt{decompose(x,tx,ty)}$$

where *decompose* is the *Decompose* predicate synthesized at step 3. Then infer:

$$\texttt{procComp(hx,ty,y)} \Leftarrow \texttt{Body} \qquad (P'_i)$$

as a property of $procComp$.

A logic algorithm $LA(procComp)$ can now be synthesized from this inferred specification by examples and properties. The synthesis of $LA(r)$ proceeds using $procComp$.

1.3.2 Illustration on sample problems

We illustrate the above methods and heuristic on the sample problems of section 1.2.

1.3.2.1 THE $EFFACE/3$ PROBLEM

The logic algorithm being synthesized for the $efface/3$ problem is what we call a *partial scan* logic algorithm: the recursion may stop scanning the induction parameter before being through. Indeed, once E has been located in L, one can already compute the final value of R without further scanning L. At step 4, the actual values of the introduced parameter TR could be determined because $efface/3$ is a deterministic problem, so no lifting of non-determinacy is required here.

According to the MSG Method, there are initially no classes, so no disjunct belongs to any class. We first consider disjunct DE_3. It can't be compatible with any class, because there are none so far. So we create a singleton class $\{DE_3\}$.

We pursue with DE_5. To see whether DE_5 is compatible with class $\{DE_3\}$, we compute the msg of their $\langle HL, TR, R\rangle$ value-tuples:

HL	TR	R	
d	$[]$	$[d]$	$msg\{DE_3\}$
i	$[k]$	$[i,k]$	DE_5
A	T	$[A\|T]$	$msg(msg\{DE_3\}, DE_5)$

Disjunct DE_5 goes into class $\{DE_3\}$, because it is compatible with that class ($[A|T]$ is constructed in terms of A and T).

We pursue with DE_6. To see whether DE_6 is compatible with class $\{DE_3, DE_5\}$,

$$
\begin{array}{lll}
\textbf{efface}(\mathbf{E},\mathbf{L},\mathbf{R}) \Leftrightarrow & & \\
 & \mathbf{L} = [\_] & \wedge\ \mathbf{L} = [\mathbf{E}]\ \wedge\ \mathbf{R} = [\,] \\
 & & \wedge\ E = a\ \wedge\ L = [a]\ \wedge\ R = [\,] \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \mathbf{E} = \mathbf{HL}\ \wedge\ \mathbf{R} = \mathbf{TL} \\
 & & \wedge\ E = b\ \wedge\ L = [b,c]\ \wedge\ R = [c]\ \wedge\ \ldots \\
 & & \quad \vee E = f\ \wedge\ L = [f,g,h]\ \wedge\ R = [g,h]\ \wedge\ \ldots \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \textbf{efface}(\mathbf{E},\mathbf{TL},\mathbf{TR}) \\
 & & \wedge\ \mathbf{R} = [\mathbf{HL}|\mathbf{TR}] \\
 & & \wedge\ HL = d\ \wedge\ TR = [\,]\ \wedge\ R = [d]\ \wedge\ \ldots \\
 & & \quad \vee HL = i\ \wedge\ TR = [k]\ \wedge\ R = [i,k]\ \wedge\ \ldots \\
 & & \quad \vee HL = m\ \wedge\ TR = [n]\ \wedge\ R = [m,n]\ \wedge\ \ldots
\end{array}
$$

Figure 1.12 $LA_6(efface)$

we compute the msg of their $\langle HL, TR, R\rangle$ value-tuples:

HL	TR	R	
A	T	$[A\|T]$	$msg\{DE_3, DE_5\}$
m	$[n]$	$[m,n]$	DE_6
A	T	$[A\|T]$	$msg(msg\{DE_3, DE_5\}, DE_6)$

Disjunct DE_6 goes into class $\{DE_3, DE_5\}$, because it is compatible with that class.

There are no other disjuncts. We have partitioned the non-minimal disjuncts into $c = 1$ equivalence class, namely $\{DE_3, DE_5, DE_6\}$. Let $pcEfface$ be the chosen instantiation of $ProcComp_1$. It is implemented by re-expression of the msg:

$$pcEfface(HL, TR, R) \Leftrightarrow R = [HL|TR].$$

This result is unfolded into the corresponding disjuncts of $LA_5(efface)$, and, after regrouping of disjuncts, $LA_6(efface)$ looks as depicted in figure 1.12.

1.3.2.2 THE $PERM/2$ PROBLEM

The logic algorithm being synthesized for the $perm/2$ problem is what we call a *total scan* logic algorithm: the recursion scans the induction parameter entirely. Indeed, all elements of L need to be visited so that they can be stuffed into other locations in P. The other mission of step 6 is to lift the non-determinacy about the actual values of parameter TP introduced at step 4 because of the non-determinism of the $perm/2$ problem.

According to the MSG Method, there are initially no classes, so no disjunct belongs to any class. We first consider disjunct DP_2, and make it a singleton class $\{DP_2\}$.

We pursue with DP_3. To see whether DP_3 is compatible with class $\{DP_2\}$, we

compute the msg of their $\langle HL, TP, P\rangle$ value-tuples:

HL	TP	P	
a	$[\,]$	$[a]$	$msg\{DP_2\}$
b	$[c]$	$[b, c]$	DP_3
A	T	$[A\|T]$	$msg(msg\{DP_2\}, DP_3)$

Disjunct DP_3 goes into class $\{DP_2\}$, because it is compatible with that class ($[A|T]$ is composed in terms of A and T).

We pursue with DP_4. To see whether DP_4 is compatible with class $\{DP_2, DP_3\}$, we compute the msg of their $\langle HL, TP, P\rangle$ value-tuples:

HL	TP	P	
A	T	$[A\|T]$	$msg\{DP_2, DP_3\}$
b	$[c]$	$[c, b]$	DP_4
A	T	$[B\|U]$	$msg(msg\{DP_2, DP_3\}, DP_4)$

Disjunct DP_4 is not compatible with class $\{DP_2, DP_3\}$, because $[B|U]$ is not composed in terms of A and T. There are no other classes, so we create a new singleton class $\{DP_4\}$.

We pursue with DP_5. To see whether DP_5 is compatible with class $\{DP_2, DP_3\}$, we compute the msg of their $\langle HL, TP, P\rangle$ value-tuples. Two alternatives arise:

HL	TP	P	
A	T	$[A\|T]$	$msg\{DP_2, DP_3\}$
d	$[e, f]$	$[d, e, f]$	DP_{5a}
A	T	$[A\|T]$	$msg(msg\{DP_2, DP_3\}, DP_{5a})$

or:

HL	TP	P	
A	T	$[A\|T]$	$msg\{DP_2, DP_3\}$
d	$[f, e]$	$[d, e, f]$	DP_{5b}
A	T	$[A\|U]$	$msg(msg\{DP_2, DP_3\}, DP_{5b})$

Disjunct DP_{5a} is compatible with class $\{DP_2, DP_3\}$, but DP_{5b} isn't. So DP_{5a} goes into class $\{DP_2, DP_3\}$.

The computations eventually identify $c = 3$ equivalence classes (see exercise 3).

HL	TP	P	
A	T	$[A\|T]$	$msg\{DP_2, DP_3, DP_{5a}, DP_{6b}\}$
A	$[B\|T]$	$[B, A\|T]$	$msg\{DP_4, DP_{7a}, DP_{9b}\}$
A	$[B, C\|T]$	$[B, C, A\|T]$	$msg\{DP_{8a}, DP_{10b}\}$

Applying heuristic 1 (there are more equivalence classes than non-minimal properties), we invoke the Synthesis Method. Let $pcPerm$ be the chosen instantiation of $ProcComp_1$. A specification by examples and properties for $pcPerm(HL, TP, P)$ has to be inferred from $LA_5(perm)$.

The inference of an example set is done by extracting all the $\langle HL, TP, P\rangle$ value-

$E(pcPerm)$	$= \{$	`pcPerm(a,[],[a])`	(ES_1)
		`pcPerm(b,[c],[b,c])`	(ES_2)
		`pcPerm(b,[c],[c,b])`	(ES_3)
		`pcPerm(d,[e,f],[d,e,f])`	(ES_4)
		`pcPerm(d,[f,e],[d,f,e])`	(ES_5)
		`pcPerm(d,[e,f],[e,d,f])`	(ES_6)
		`pcPerm(d,[e,f],[e,f,d])`	(ES_7)
		`pcPerm(d,[f,e],[f,d,e])`	(ES_8)
		`pcPerm(d,[f,e],[f,e,d])` $\}$	(ES_9)
$P(pcPerm)$	$= \{$	`pcPerm(X,[],[X])`	(PS_1)
		`pcPerm(X,[Y],[X,Y])`	(PS_2)
		`pcPerm(X,[Y],[Y,X])` $\}$	(PS_3)

Figure 1.13 Derived example and property sets for $pcPerm(E, L, R)$

tuples from the non-minimal disjuncts of $LA_5(perm)$. If there are several possible values for TP, we extract the one that led to compatibility during the MSG Method. The result is shown in figure 1.13. Note that example ES_5 is a variant of ES_4, that ES_8 is a variant of ES_6, and that ES_9 is a variant of ES_7.

The inference of a property set is here performed as follows. For every property:

$$\texttt{perm(x}_2\texttt{,y}_2\texttt{)} \Leftarrow \texttt{Body} \qquad (PP_i)$$

find a variant:

$$\texttt{perm(x}_1\texttt{,y}_1\texttt{)} \qquad (EP_j/PP_j)$$

of an example or body-less property, such that:

$$\texttt{x}_2 = \texttt{[h|y}_1\texttt{]}$$

Infer:

$$\texttt{pcPerm(h,y}_1\texttt{,y}_2\texttt{)} \Leftarrow \texttt{Body} \qquad (PS_i)$$

as a property of $pcPerm$.

In our case, the pairs $\langle PP_1, EP_1\rangle$, $\langle PP_2, PP_1\rangle$, and $\langle PP_3, PP_1\rangle$ infer the properties shown in figure 1.13. The informal specification is that `pcPerm(E,L,R)` succeeds iff list `R` is list `L` with term `E` stuffed into a random location. But, in order to prevent redundant solutions, element `E` should only be stuffed into some location of `L` that precedes its own first occurrence (if any) in `L`. The following new version of PP_3 achieves this:

$$\texttt{perm([X,Y],[Y,X])} \Leftarrow \texttt{X} \neq \texttt{Y} \qquad (PP'_3)$$

because PS_3 then reads:

$$\texttt{pcPerm(X,[Y],[Y,X])} \Leftarrow \texttt{X} \neq \texttt{Y} \qquad (PS'_3)$$

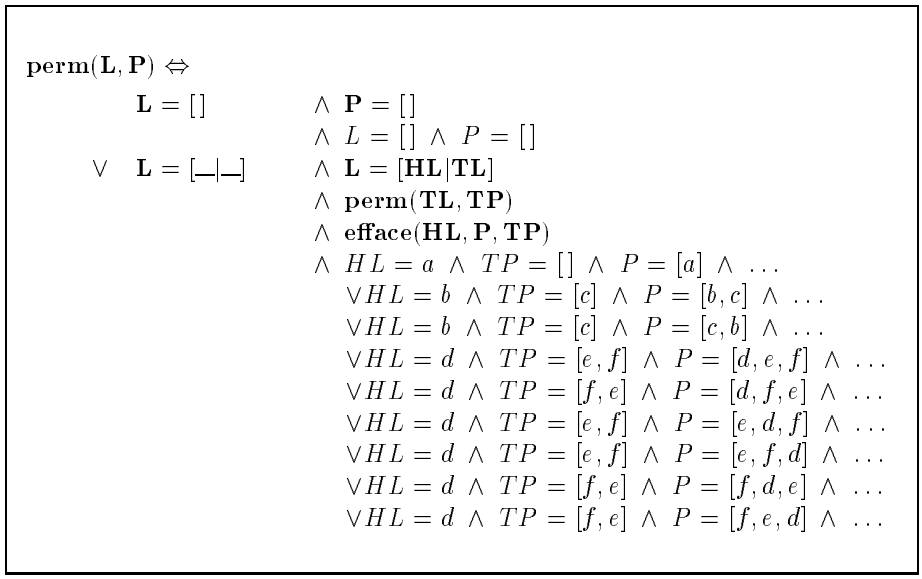

Figure 1.14 $LA_6(perm)$

Note that when replacing all occurrences of the empty list [] in $\{PS_1, PS_2, PS'_3\}$ by a variable T, this property set collapses into the one of the $efface/3$ problem, where the second and third parameters have been exchanged. When taking into account the above-noted variants in $E(pcPerm)$, then the example set also collapses into the one of the $efface/3$ problem, where the second and third parameters have been exchanged. The new synthesis is thus, up to renaming of the predicate and re-ordering of the parameters, the same as the one of the $efface/3$ problem. Hence $LA_6(perm)$ looks as depicted in figure 1.14.

1.3.2.3 THE $FIRSTSEQ/3$ PROBLEM

The logic algorithm being synthesized for the $firstSeq/3$ problem a partial scan logic algorithm: the recursion may stop scanning L before it is through. Indeed, once F has been identified in L, one can already compute the final value of S without further scanning L. At step 4, the actual values of the introduced parameters TF and TS could be determined because $firstSeq/3$ is a deterministic problem, so no lifting of non-determinacy is required here.

According to the MSG Method, there are initially no classes, so no disjunct belongs to any class. We first consider disjunct DF_3, and create a singleton class $\{DF_3\}$.

We pursue with DF_4. To see whether DF_4 is compatible with class $\{DF_3\}$, we

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{firstSeq}(\mathbf{L},\mathbf{F},\mathbf{S}) \Leftrightarrow} \\
 & \mathbf{L} = [\_] & \wedge\ \mathbf{F} = \mathbf{L}\ \wedge\ \mathbf{S} = [\,] \\
 & & \wedge\ L = [a]\ \wedge\ F = [a]\ \wedge\ S = [\,] \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \mathbf{TL} = [\mathbf{HTL}|\_]\ \wedge\ \mathbf{HL} \neq \mathbf{HTL} \\
 & & \wedge\ \mathbf{F} = [\mathbf{HL}]\ \wedge\ \mathbf{S} = \mathbf{TL} \\
 & & \wedge\ L = [b,c,d,e]\ \wedge\ F = [b]\ \wedge\ S = [c,d,e] \\
 & & \wedge\ HL = b\ \wedge\ TL = [c,d,e] \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \mathbf{firstSeq}(\mathbf{TL},\mathbf{TF},\mathbf{TS}) \\
 & & \wedge\ \mathbf{F} = [\mathbf{HL}|\mathbf{TF}]\ \wedge\ \mathbf{S} = \mathbf{TS} \\
 & & \wedge\ HL = f\ \wedge\ TF = [f]\ \wedge\ TS = [g,h]\ \wedge\ \ldots \\
 & & \quad \vee HL = i\ \wedge\ TF = [i,i]\ \wedge\ TS = [j]\ \wedge\ \ldots
\end{array}
$$

Figure 1.15 $LA6(firstSeq)$

compute the msg of their $\langle HL, TF, TS, F, S\rangle$ value-tuples:

HL	TF	TS	F	S	
f	$[f]$	$[g,h]$	$[f,f]$	$[g,h]$	$msg\{DF_3\}$
i	$[i,i]$	$[j]$	$[i,i,i]$	$[j]$	DF_4
A	$[A\|T]$	$[B\|U]$	$[A,A\|T]$	$[B\|U]$	$msg\{DF_3, DF_4\}$

Disjunct DF_4 is compatible with class $\{DF_3\}$, because $[A, A|T]$ is composed in terms of A, A, and T, and because $[B|U]$ is composed in terms of A, B, and U. Thus DF_4 goes into that class.

There are no other disjuncts. We have partitioned the non-minimal disjuncts into $c = 1$ equivalence class, namely $\{DF_3, DF_4\}$. Let $pcFirstSeq$ be the chosen instantiation of $ProcComp_1$. It is implemented by re-expression of the msg:

$$pcFirstSeq(HL, TF, TS, F, S) \Leftrightarrow F = [HL|TF] \wedge S = TS.$$

This result is unfolded into the corresponding disjuncts of $LA_5(firstSeq)$, and, after regrouping of disjuncts, $LA_6(firstSeq)$ looks as depicted in figure 1.15.

1.4 SYNTHESIS OF THE $DISCRIMINATE_K$ (STEP 7)

In the non-minimal recursive case, the $Discriminate_k(\mathbf{HX}, \mathbf{TX}, Y)$ procedures perform some tests so as to ensure that the parameters effectively should be processed and composed as in class $|k|$. In this section, we first formally present the objective and methods of step 7. Then we synthesize $Discriminate_k$ operators for the sample problems given in section 1.2.

1.4.1 Formalization: objective and methods

At step 7, the aim is to transform $LA_6(r)$ into $LA_7(r)$ such that it fits the following schema:

$$
\begin{array}{lll}
r(X,Y) \Leftrightarrow & & \\
 & minimal(X) & \wedge\ solve(X,Y) \\
 & & \wedge\ \bigvee_{1\le i\le b} X = x_i \wedge Y = y_i \\
\vee \bigvee_{1\le k\le c} & nonMinimal(X) & \wedge\ decompose(X, \mathbf{HX}, \mathbf{TX}) \\
 & & \wedge\ Discriminate_k(\mathbf{HX}, \mathbf{TX}, Y) \\
 & & \wedge\ \mathbf{r}(\mathbf{TX}, \mathbf{TY}) \\
 & & \wedge\ procComp_k(\mathbf{HX}, \mathbf{TY}, Y) \\
 & & \wedge\ \bigvee_{i\in|k|} X = x_i \wedge Y = y_i \\
 & & \quad \wedge HX = hx_i\ \wedge\ TX = tx_i\ \wedge\ TY = ty'_i
\end{array}
$$

This objective is achieved by two consecutive tasks:

i) synthesis of specialized logic algorithms of the $Discriminate_k$;
ii) generalization of these specialized logic algorithms of the $Discriminate_k$.

They are performed by a *Proofs-as-Programs Method* and a *Generalization Method*, respectively. We discuss these methods in turn.

1.4.1.1 THE PROOFS-AS-PROGRAMS METHOD

The Proofs-as-Programs Method extracts specialized implementations of the $Discriminate_k$ from the proofs that the heads of the properties are logical consequences of the recursive, non-minimal disjuncts. Indeed, properties contain explicit information that has not yet been synthesized into these disjuncts. This is thus the right moment to use this information.

When considering property P_i, let $\mathcal{T}_i$ be a theory composed of:

i) the recursive, non-minimal disjuncts of the generalization $\Gamma(LA_6(r))$;
ii) the example set $E(r)$;
iii) the property set $P(r) \setminus \{P_i\}$;
iv) logic algorithms of the primitive predicates, such as $LA(=)$.

The Proofs-as-Programs Method first attempts to prove that the head of property P_i is a logical consequence of theory $\mathcal{T}_i$. The body of property P_i is then used during the discriminant extraction.

These proofs can be done by a slight modification of SLD resolution (see section ??). A clausal version of $\mathcal{T}_i$ has first to be generated: this is straightforward due to our restriction to logic algorithms whose bodies are in disjunctive normal form. Examples and properties are already in clausal form.

The initial goal is the head of property P_i.

The *computation rule* satisfies the following condition: never select an atom with predicate r/n if there are atoms with primitive predicates.

The search rule is as follows:

- if the selected atom has a primitive predicate, then it is resolved according to its semantics;

- the root of the proof tree is resolved using the clauses generated from $\Gamma(LA_6(r))$;
- if the selected atom (in a non-root node) has predicate r/n, then it is resolved using the clauses generated from $\mathcal{T}_i \setminus \{\Gamma(LA_6(r))\}$.

In the resolution of the root, we use $\Gamma(LA_6(r))$ rather than the examples or properties because that wouldn't make sense: we are trying to prove the logic algorithm correct wrt its specification, not to prove the specification consistent. In the resolution of atoms with predicate r/n, we use the examples and other properties rather than $\Gamma(LA_6(r))$ because that wouldn't make sense either: $\Gamma(LA_6(r))$ is still incorrect. Our restriction to properties that are non-recursive comes in handy here: there is no further nesting of goals in r/n.

A derivation of the head of property P_i succeeds iff it ends in the empty clause. Let k be the number of the (recursive, non-minimal) disjunct of $\Gamma(LA_6(r))$ whose generated clause was used in the initial resolution step. Let σ_{ki} be the corresponding computed answer substitution. We instantiate $Discriminate_k$ by $discriminate_k$ such that it is partly defined by the following clause:

$$\texttt{discriminate}_k\texttt{(HX,TX,y)}\sigma_{ki} \leftarrow \texttt{Body}_i\sigma_{ki}$$

where $\texttt{Body}_i$ is the body of property P_i, and where y is the value of Y in the head of P_i.

Note that this is not like classical program extraction from proofs (compare with [BSW90, Fri90, MW79]), since the program is here extracted from the unique final result of the proof, rather than on-the-fly (or a posteriori) from multiple intermediate proof-steps.

A derivation *fails* iff it doesn't succeed. Nothing is done in that case. Failure is of course not always detectable, since an infinite derivation may occur. In practice, success is approximated by limiting the size (or the CPU time) of a derivation.

The proof of a property set *succeeds* iff every property has at least one successful derivation. The revealed clauses of discriminants are then assembled into logic algorithms:

$$\begin{aligned} &discriminate_k(\mathbf{HX}_k, \mathbf{TX}_k, Y_k) \Leftrightarrow \\ &\qquad \bigvee_{i \in \mathcal{S}(k)} (\mathbf{HX}_k = \mathbf{HX} \ \wedge \ \mathbf{TX}_k = \mathbf{TX} \ \wedge \ Y_k = y \ \wedge \ Body_i)\sigma_{ki} \end{aligned}$$

where $\mathcal{S}(k)$ is the finite, possibly empty, set of indices of the properties whose proofs revealed clauses of $discriminate_k$. If $\mathcal{S}(k)$ is empty, then the corresponding discriminant always evaluates to $true$.

The proof of a property set *fails* iff some property has no successful derivation. In such a case, the Proofs-as-Programs Method does not synthesize the discriminants. Previous steps of the synthesis have then to be reconsidered.

1.4.1.2 THE GENERALIZATION METHOD

Since the properties only embody fragmentary information, the discriminants obtained so far are too specialized. The Generalization Method applies some heuristics, and postulates that the resulting discriminants are the intended ones.

Here are some valuable heuristics:

Heuristic 2 *If the value of some* $\mathbf{TX}_k$ *or* Y_k *is not changed by the computed answer substitution, then this parameter is irrelevant and its value can be generalized to the anonymous variable.*

Heuristic 3 *The parameter* $\mathbf{TX}_k$ *should be a compound induction parameter of* $LA(discriminate_k)$*: if necessary, some values of the* $\mathbf{TX}_k$ *should be generalized so that the* $\mathbf{TX}_k$ *range across their entire domain.*

The Generalization Method is always applicable.

1.4.2 Illustration on sample problems

We illustrate the above methods and heuristics on the sample problems of section 1.2.

1.4.2.1 THE $EFFACE/3$ PROBLEM

The Proofs-as-Programs Method attempts to prove, by SLD-resolution, that the head of property PE_i is a logical consequence of theory $\mathcal{TE}_i$.

Starting with PE_1, we have:[5]

$$\leftarrow \underline{efface(X,[X|T],T)}$$

$$\text{disjunct 3 of } \Gamma(LA_6(efface)) \;\Big|\; \{\}$$

$$\leftarrow \underline{[X|T]=[HL|TL]},\ efface(X,TL,TR),\ T=[HL|TR]$$

$$LA(=) \;\Big|\; \{HL/X, TL/T\}$$

$$\leftarrow efface(X,T,TR),\ \underline{T=[X|TR]}$$

$$LA(=) \;\Big|\; \{T/[X|TR]\}$$

$$\leftarrow \underline{efface(X,[X|TR],TR)}$$

The last goal doesn't unify with the head of any clause in $\mathcal{TE}_1$: the derivation fails. The head of property PE_1 is not a logical consequence of theory $\mathcal{TE}_1$. So nothing is done to disjunct 3.

[5] In a goal, the underlined atom(s) are the selected atom(s) for the next resolution step(s).

We pursue with PE_2:

$$\leftarrow \underline{efface(X,[Y,X|T],[Y|T])}$$

$$\text{disjunct 3 of } \Gamma(LA_6(efface)) \;\Big|\; \{\}$$

$$\leftarrow \underline{[Y,X|T]=[HL|TL]},\ efface(X,TL,TR),\ [Y|T]=[HL|TR]$$

$$LA(=) \;\Big|\; \{HL/Y, TL/[X|T]\}$$

$$\leftarrow efface(X,[X|T],TR),\ \underline{[Y|T]=[Y|TR]}$$

$$LA(=) \;\Big|\; \{TR/T\}$$

$$\leftarrow \underline{efface(X,[X|T],T)}$$

$$PE_1 \;\Big|\; \{\}$$

$$\square$$

The head of property PE_2 is a logical consequence of theory $\mathcal{TE}_2$. A specialized discriminant for disjunct 3 is partly defined as follows:

$$\texttt{discEfface(HL,TL,X,[X|T])}\sigma \leftarrow \texttt{X} \neq \texttt{Y}\sigma$$

where σ is the computed answer substitution of the above SLD-derivation. This yields:

$$\texttt{discEfface(Y,[X|T],X,[X|T])} \leftarrow \texttt{X} \neq \texttt{Y}$$

There is no other property. There are no alternative derivations.

The Generalization Method applies heuristic 2 to decide that the values of the second and fourth parameters of the discriminant are irrelevant in this case. After generalization of the implication into an equivalence, plus renaming of some variables, the discriminant reads:

$$discEfface(HL,TL,E,R) \Leftrightarrow E \neq HL \wedge TL = \_ \wedge R = \_.$$

This result is simplified and then unfolded into disjunct 3, so $LA_7(efface)$ looks as depicted in figure 1.16.

1.4.2.2 THE $PERM/2$ PROBLEM

At step 6, the Synthesis Method has been invoked for the $perm/2$ problem, because there is only one equivalence class. The properties of $perm/2$ have been inherited by

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{efface}(\mathbf{E},\mathbf{L},\mathbf{R}) \Leftrightarrow} \\
 & \mathbf{L} = [\_] & \wedge\ \mathbf{L} = [\mathbf{E}]\ \wedge\ \mathbf{R} = [\,] \\
 & & \wedge\ E = a\ \wedge\ L = [a]\ \wedge\ R = [\,] \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \mathbf{E} = \mathbf{HL}\ \wedge\ \mathbf{R} = \mathbf{TL} \\
 & & \wedge\ E = b\ \wedge\ L = [b,c]\ \wedge\ R = [c]\ \wedge\ \ldots \\
 & & \quad \vee E = f\ \wedge\ L = [f,g,h]\ \wedge\ R = [g,h]\ \wedge\ \ldots \\
\vee & \mathbf{L} = [\_,\_|\_] & \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
 & & \wedge\ \mathbf{E} \neq \mathbf{HL} \\
 & & \wedge\ \mathbf{efface}(\mathbf{E},\mathbf{TL},\mathbf{TR}) \\
 & & \wedge\ \mathbf{R} = [\mathbf{HL}|\mathbf{TR}] \\
 & & \wedge\ HL = d\ \wedge\ TR = [\,]\ \wedge\ R = [d]\ \wedge\ \ldots \\
 & & \quad \vee HL = i\ \wedge\ TR = [k]\ \wedge\ R = [i,k]\ \wedge\ \ldots \\
 & & \quad \vee HL = m\ \wedge\ TR = [n]\ \wedge\ R = [m,n]\ \wedge\ \ldots
\end{array}
$$

Figure 1.16 $LA_7(efface)$

the inferred subproblem, so no discriminant is needed. Hence $LA_7(perm)$ is identical to $LA_6(perm)$.

1.4.2.3 THE $FIRSTSEQ/3$ PROBLEM

The Proofs-as-Programs Method attempts to prove, by SLD-resolution, that the head of property PF_i is a logical consequence of theory $\mathcal{TF}_i$.

We start with PF_1:

$$
\begin{array}{c}
\leftarrow \underline{firstSeq([X],[X],[\,])} \\
\text{disjunct 3 of } \Gamma(LA_6(firstSeq)) \ \Big|\ \{\,\} \\
\leftarrow \underline{[X] = [HL|TL]},\ firstSeq(TL,TF,TS),\ \underline{[X] = [HL|TF]},\ \underline{[\,] = TS} \\
LA(=) \ \Big|\ \{HL/X, TL/[\,], TF/[\,], TS/[\,]\} \\
\leftarrow \underline{firstSeq([\,],[\,],[\,])}
\end{array}
$$

The last goal doesn't unify with the head of any clause in $\mathcal{TF}_1$: the derivation fails. The head of property PF_1 is not a logical consequence of theory $\mathcal{TF}_1$. So nothing is done to disjunct 3.

We pursue with PF_3:

$$\leftarrow \underline{firstSeq([X,Y|T],[X],[Y|T])}$$

$$\text{disjunct 3 of } \Gamma(LA_6(firstSeq)) \;\Big|\; \{\}$$

$$\leftarrow \underline{[X,Y|T] = [HL|TL]},\ firstSeq(TL,TF,TS),\ \underline{[X] = [HL|TF]},\ \underline{[Y|T] = TS}$$

$$LA(=) \;\Big|\; \{HL/X, TL/[Y|T], TF/[], TS/[Y|T]\}$$

$$\leftarrow \underline{firstSeq([Y|T],[],[Y|T])}$$

The last goal doesn't unify with the head of any clause in $\mathcal{TF}_3$: the derivation fails. The head of property PF_3 is not a logical consequence of theory $\mathcal{TF}_3$. So nothing is done to disjunct 3.

We pursue with PF_2:

$$\leftarrow \underline{firstSeq([X,X],[X,X],[])}$$

$$\text{disjunct 3 of } \Gamma(LA_6(firstSeq)) \;\Big|\; \{\}$$

$$\leftarrow \underline{[X,X] = [HL|TL]},\ firstSeq(TL,TF,TS),\ \underline{[X,X] = [HL|TF]},\ \underline{[] = TS}$$

$$LA(=) \;\Big|\; \{HL/X, TL/[X], TF/[X], TS/[]\}$$

$$\leftarrow \underline{firstSeq([X],[X],[])}$$

$$PF_1 \;\Big|\; \{\}$$

$$\square$$

The head of property PF_2 is a logical consequence of theory $\mathcal{TF}_2$. A specialized discriminant for disjunct 3 is partly defined as follows:

$$\texttt{discFirstSeq(HL,TL,[X,X],[])}\sigma \leftarrow \texttt{true}\sigma$$

where σ is the computed answer substitution of the above SLD-derivation. This yields:

```
discFirstSeq(X,[X],[X,X],[])
```

There is no other property. There are no alternative derivations.

The Generalization Method applies heuristic 2 to decide that the values of the third and fourth parameters are irrelevant in this case. It applies heuristic 3 to generalize the TL parameter into non-empty lists. After generalization of the implication into an

$$
\begin{array}{l}
\mathbf{firstSeq}(\mathbf{L},\mathbf{F},\mathbf{S}) \Leftrightarrow \\
\qquad \mathbf{L} = [\_] \qquad\quad \wedge\ \mathbf{F} = \mathbf{L} \wedge\ \mathbf{S} = [\,] \\
\qquad\qquad\qquad\quad \wedge\ L = [a] \wedge\ F = [a] \wedge\ S = [\,] \\
\quad \vee \quad \mathbf{L} = [\_,\_|\_] \quad \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{TL} = [\mathbf{HTL}|\_] \wedge\ \mathbf{HL} \neq \mathbf{HTL} \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{F} = [\mathbf{HL}] \wedge\ \mathbf{S} = \mathbf{TL} \\
\qquad\qquad\qquad\quad \wedge\ L = [b,c,d,e] \wedge\ F = [b] \wedge\ S = [c,d,e] \\
\qquad\qquad\qquad\quad \wedge\ HL = b \wedge\ TL = [c,d,e] \\
\quad \vee \quad \mathbf{L} = [\_,\_|\_] \quad \wedge\ \mathbf{L} = [\mathbf{HL}|\mathbf{TL}] \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{TL} = [\mathbf{HTL}|\_] \wedge\ \mathbf{HL} = \mathbf{HTL} \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{firstSeq}(\mathbf{TL},\mathbf{TF},\mathbf{TS}) \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{F} = [\mathbf{HL}|\mathbf{TF}] \wedge\ \mathbf{S} = \mathbf{TS} \\
\qquad\qquad\qquad\quad \wedge\ HL = f \wedge\ TF = [f] \wedge\ TS = [g,h] \wedge\ \ldots \\
\qquad\qquad\qquad\qquad \vee HL_1 = i \wedge\ TF = [i,i] \wedge\ TS = [j] \wedge\ \ldots
\end{array}
$$

Figure 1.17 $LA_7(firstSeq)$

equivalence, plus renaming of some variables, the discriminant reads:

$$discFirstSeq(HL,TL,F,S) \Leftrightarrow TL = [HTL|\_] \wedge HL = HTL \wedge F = \_ \wedge S = \_ .$$

This result is simplified and then unfolded into disjunct 3, so $LA_7(firstSeq)$ looks as depicted in figure 1.17.

1.5 SYNTACTIC GENERALIZATION (STEP 8)

In this section, we first formally present the objective and method of step 8. Then we compute the final generalizations for the sample problems given in section 1.2.

1.5.1 Formalization: objective and method

At step 8, the objective is to transform $LA_7(r)$ into $LA_8(r)$ that looks like:

$$
\begin{array}{l}
r(X,Y) \Leftrightarrow \\
\qquad\qquad\quad minimal(X) \qquad\quad \wedge\ solve(X,Y) \\
\quad \vee \bigvee_{1 \leq k \leq c} \quad nonMinimal(X) \qquad \wedge\ decompose(X,\mathbf{HX},\mathbf{TX}) \\
\qquad\qquad\qquad\qquad\quad \wedge\ discriminate_k(\mathbf{HX},\mathbf{TX},Y) \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathbf{r}(\mathbf{TX},\mathbf{TY}) \\
\qquad\qquad\qquad\qquad\quad \wedge\ procComp_k(\mathbf{HX},\mathbf{TY},Y)
\end{array}
$$

All predicate variables of the divide-and-conquer schema have already been instantiated until step 7, and we have used all the information contained in the specification:

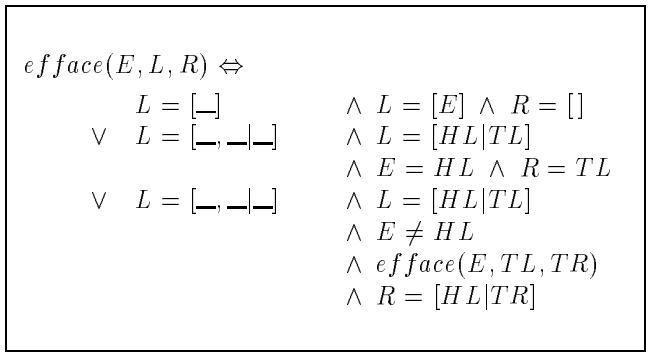

Figure 1.18 $LA_8(efface)$

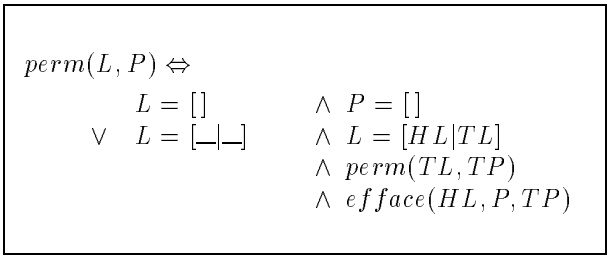

Figure 1.19 $LA_8(perm)$

- examples are injected at step 1 and are trailed along all subsequent steps in the form of equality atoms, so that they be present when needed;
- examples and properties are used at step 4 to deductively infer values of the **TY**;
- examples and properties are used at step 6 to infer specifications of subproblems;
- properties are used at steps 5 and 7 to infer discriminants.

So we may consider the synthesis task finished. We postulate that $LA_8(r)$ is $\Gamma(LA_7(r))$.

1.5.2 Illustration on sample problems

The final generalizations for the sample problems given in section 1.2 are depicted in figures 1.18, 1.19, and 1.20, respectively. Note that they can be proven to be equivalent to the sample logic algorithms listed in figures 1.4, 1.7, and 1.10, respectively.

1.6 CONCLUSION

In this last section, we present the framework, results, and contributions of this research, and mention some related research, as well as future research.

$$
\begin{array}{lll}
firstSeq(L,F,S) \Leftrightarrow & & \\
\quad & L = [\_] & \wedge\ F = L\ \wedge\ S = [\,] \\
\vee & L = [\_,\_|\_] & \wedge\ L = [HL|TL] \\
 & & \wedge\ TL = [HTL|\_]\ \wedge\ HL \neq HTL \\
 & & \wedge\ F = [HL]\ \wedge\ S = TL \\
\vee & L = [\_,\_|\_] & \wedge\ L = [HL|TL] \\
 & & \wedge\ TL = [HTL|\_]\ \wedge\ HL = HTL \\
 & & \wedge\ firstSeq(TL,TF,TS) \\
 & & \wedge\ F = [HL|TF]\ \wedge\ S = TS
\end{array}
$$

Figure 1.20 $LA_8(firstSeq)$

1.6.1 Framework, results, and contributions

This research was partly led within the framework of the *Folon* research project (Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium) whose objective is twofold. First, it aims at elaborating a methodology of logic program development, such as described in [Dev90]. Second, it aims at designing an integrated set of tools supporting this methodology. Our research tackles the aspects of logic program synthesis from examples and properties.

The main results of our work on logic program synthesis so far are the definition of a synthesis calculus ([FD91, FD93]), the identification of a particular synthesis mechanism ([Fle91]), and the development of methods for each of its steps ([Fle91]).

One of the originalities of our approach is the combination of examples with properties, so as to cope with some classical difficulties of example-based synthesis.

1.6.2 Related research

Pointers to related research in program synthesis have been given throughout the chapter, and we have already stressed in detail how our approach differs from the state of the art. The use of schemata is also advocated in [Smi85, DB89, GH89, Tin90] (divide-and-conquer), [Smi88] (global search), and others, although sometimes in different contexts (e.g. programming tutors/assistants). An early study of the concept of most specific generalization is [Plo70].

1.6.3 Future research

In the future, we plan to pursue research on the following aspects:

Development of a proof-of-concept implementation (in Prolog) of the synthesis mechanism, called SYNAPSE (SYNthesis of logic Algorithms from PropertieS and Examples). This should allow the identification of points of interaction with the specifier so that we can better cope with incompleteness: wherever inductive inference is used, the specifier should be able to give his feedback. It is important to keep this dialogue

easy, i.e. a question-and-answer method asking for the classification of ground atoms as examples or counter-examples seems to be an appropriate choice.

Incorporation of counter-examples in the specifications, the general synthesis strategy, and the synthesis mechanism. Indeed, negative information is quite useful in avoiding over-generalization during inductive inference.

Formulation of a methodology of choosing examples and properties. It is important to guide the specifier towards choosing "good" examples and properties. This reduces interaction with the specifier, and results thus in highly automated synthesis.

1.7 EXERCISES

1. Give a sample logic program fitting the divide-and-conquer schema, where the instances of the $NonMinimal$ and the $Decompose$ predicates are different.
2. Extend the schema of figure 1.1 to n-ary relations.
3. Perform the remaining msg computations of subsection 1.3.2.2.
4. Perform the Proofs-as-Program Method to the $perm/2$ problem. Show that the obtained discriminants are redundant with the existing literals of $LA_6(perm)$.
5. Show that $LA_8(efface)$, $LA_8(perm)$, and $LA_8(firstSeq)$ are logically equivalent to (i.e. have the same models than) the logic algorithms listed in figures 1.4, 1.7, and 1.10, respectively.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge many insightful discussions with Prof. B. Le Charlier (Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium). Thanks also to the anonymous reviewers for their constructive comments on this paper. Many thanks to Jean-Marie Jacquet for his careful editing. Parts of the results presented here were found while the first author was on leave at Duke University (NC, USA). The first author is supported by the Government of Luxembourg, Ministry of Scientific Research and Cultural Affairs, Grant BFR 92/017.

REFERENCES

[Bie84] A.W. Biermann. Dealing with Search. In A.W. Bierman, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 375–392. Macmillan, New York, NY, USA, 1984.

[Bie92] A.W. Biermann. Automatic Programming. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, Wiley Interscience Publication, pages 59–83. John Wiley & Sons, New York, NY, USA, second, extended edition, 1992.

[BS79] A.W. Biermann and D.R. Smith. A Production Rule Mechanism for Generating LISP Code. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(5):260–276, 1979.

[BSW90] A. Bundy, A. Smaill, and G. Wiggins. The Synthesis of Logic Programs from Inductive Proofs. In J.W. Lloyd, editor, *Symposium on Computational Logic*, Esprit Basic Research Series, pages 135–149. Springer-Verlag, Heidelberg, Germany, 1990.

[Cla81] K.L. Clark. The Synthesis and Verification of Logic Programs. Research Report DOC-81/36, Imperial College, London, United Kingdom, September 1981.

[DB89] Y. Deville and J. Burnay. Generalization and Program Schemata: A Step Towards Computer-Aided Construction of Logic Programs. In E.L. Lusk and R. Overbeek, editors, *Proc. North American Conference on Logic Programming*, Series in Logic Programming, pages 409–425, Cleveland, USA, October 1989. The MIT Press.

[Dev90] Y. Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming. Addison-Wesley, Wokingham, United Kingdom, 1990.

[FD91] P. Flener and Y. Deville. Towards Stepwise, Schema-Guided Synthesis of Logic Programs. In K. Lau and T. Clement, editors, *Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing Series, pages 46–64, Manchester, United Kingdom, July 1991. Springer-Verlag.

[FD93] P. Flener and Y. Deville. Logic Program Synthesis from Incomplete Specifications. *Journal of Symbolic Computation: Special Issue on Automatic Programming*, 1993. To appear.

[Fle91] P. Flener. Towards Programming by Examples and Properties. Research Report CS-1991-09, Duke University, Durham, NC, USA, March 1991.

[Fle93] P. Flener. *Algorithm Synthesis from Incomplete Specifications*. PhD thesis, Unité d'Informatique, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1993. Forthcoming.

[Fri90] L. Fribourg. Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, Series in Logic Programming, pages 685–699, Jerusalem, Israel, June 1990. The MIT Press.

[GH89] T.S. Gegg-Harrison. Basic Prolog Schemata. Research Report CS-1989-20, Duke University, Durham, NC, USA, September 1989.

[Han80] Å. Hansson. *A Formal Development of Programs*. PhD thesis, Department of Computer Science, University of Stockholm, Stockholm, Sweden, 1980.

[Hog81] C.J. Hogger. Derivation of Logic Programs. *Journal of the ACM*, 28(2):372–392, 1981.

[LP90] K. Lau and S. Prestwich. Top-Down Synthesis of Recursive Logic Procedures from First-Order Logic Specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, Series in Logic Programming, pages 667–684, Jerusalem, Israel, June 1990. The MIT Press.

[Man74] Z. Manna. *Mathematical Theory of Computation*. Computer Science Series. McGraw-Hill, New York, NY, USA, 1974.

[Mug92] S. Muggleton, editor. *Inductive Logic Programming*, volume 38 of *APIC Series*. Academic Press, London, United Kingdom, 1992.

[MW79] Z. Manna and R. Waldinger. Synthesis: Dreams ⇒ Programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[Plo70] G.D. Plotkin. A Note on Inductive Generalization. *Machine Intelligence*, 5:153–163, 1970.

[Sha82] E. Shapiro. *Algorithmic Program Debugging*. The ACM Distinguished Dissertation Series. The MIT Press, Cambridge, MA, USA, 1982.

[Smi85] D.R. Smith. Top-down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[Smi88] D.R. Smith. The Structure and Design of Global Search Algorithms. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, USA, July 1988.

[Sum77] P. Summers. A Methodology for LISP Program Construction from Examples. *Journal of the ACM*, 24(1):161–175, 1977.

[Tin90] N.L. Tinkham. *Induction of Schemata for Program Synthesis*. PhD thesis, Duke University, Durham, NC, USA, 1990.